Met Office

Science & Technology Facilities Council

NERC SCIENCE OF THE ENVIRONMENT

GungHo Phase 1

Computational Science Recommendations

Forecasting Research
Technical Report No: 587

November 2013

Rupert Ford, Matthew Glover, David Ham,
Chris Maynard, Stephen Pickles,
Graham Riley

Contents

1 Introduction **3**

1.1 Future architectures . . . 3

1.2 Consequences for atmospheric models . . . 4

1.3 Design considerations for a dynamical core . . . 4

2 Programming Language **5**

3 Quasi-uniform meshes **5**

4 Proposed Software Architecture **8**

4.1 Driver layer . . . 9

4.2 Driver/Algorithm interface . . . 10

4.3 Algorithm layer . . . 10

4.4 Algorithm to PSy interface . . . 11

4.5 PSy layer . . . 14

4.6 PSy to Kernel interface . . . 14

4.7 Kernel layer . . . 15

4.8 Infrastructure layer . . . 16

5 Data model **17**

5.1 Data models for Gung Ho meshes . . . 18

5.1.1 Horizontally unstructured columnar meshes . . . 18

5.1.2 Choice of discrete function space . . . 18

5.2 Key data objects . . . 18

5.3 Mesh topology . . . 19

5.3.1 Extending the mesh topology to incorporate columns . . . 19

5.3.2 Implementing the topology . . . 20

5.3.3 The mesh topology provides the iteration spaces . . . 21

5.4 Function spaces . . . 22

5.4.1 Local function space information. . . . 22

5.4.2 Global function space information . . . 23

5.4.3 Implementing function spaces . . . 25

5.5 Field objects . . . 25

5.6 Geometry . . . 25

5.7 Addressing . . . 25

5.8 Parallelism . . . 27

5.8.1 Distributed memory parallelism . . . 27

5.8.2 Colouring and threading . . . 28

© Joint Crown and NERC Copyright 2013

6 Conclusion 28

7 Glossary 29

© Joint Crown and NERC Copyright 2013

1 Introduction

The GungHo project is a joint project between the Met Office and NERC funded under the Joint Weather and Climate Research Programme (JWCRP) and is being developed in association with STFC Daresbury. It was initiated with the expectation that the current dynamical core (New Dynamics) and next generation dynamical core (ENDGame) will not scale to the levels of parallelism that are predicted to be required to make effective use of supercomputers from around 2019 onwards. 2019 is when the GungHo dynamical core is planned to be ready for full scientific evaluation and 2022 is when the dynamical core is planned to go operational at the Met Office.

This document presents recommendations from phase 1 of the GungHo Computational Science workpackage. These recommendations are therefore targeted at the GungHo dynamical core (and a number of the recommendations will also affect the coding of the Physics routines as these are typically closely linked to the dynamical core). They are informed by Met Office requirements described in Glover (2013). Whilst it is intended that the recommendations would also benefit other Met Office models and influence the new emerging Unified Model infrastructure, these models and infrastructure, external to the GungHo dynamical core, are considered out of scope.

Author affiliations are: Rupert Ford and Stephen Pickles: STFC Daresbury; David Ham: Imperial College, London; Graham Riley: University of Manchester; Matthew Glover and Chris Maynard: The Met Office.

1.1 Future architectures

While there is still much uncertainty about what future architectures will look like in 2018 and beyond, it is clear that the trends towards shared-memory cluster nodes with an ever-increasing number of cores, stagnant or decreasing clock frequency, and decreasing memory per core, are certain to continue. This is because the design of new silicon is constrained by the imperative to minimize power consumption. Furthermore, we expect at best incremental improvements in instruction level parallelism within the execution units (SSE, AVX, and so on) on commodity processors, so that single core performance may even decrease. Consequently, simulation codes must be able to exploit ever increasing numbers of cores just to deal with current workloads on future systems, let alone achieve the next order-of-magnitude resolution improvements demanded by scientific ambitions.

The arguments (and evidence) in favour of introducing an additional thread-based layer of parallelism into an MPI code are more compelling than ever. Exploiting the shared memory within a cluster node helps to reduce the cost of communications thereby improving scalability, and to alleviate the impact of decreasing memory per core by reducing the need to store redundant copies in memory. On many architectures, it is also possible to hide much of the effect of memory latency by oversubscribing execution units with threads.

© Joint Crown and NERC Copyright 2013

1.2 Consequences for atmospheric models

Extrapolating from the Top500, the first exascale machines are likely to appear around 2018. Assuming that an individual thread can run at a peak rate of 1Gflop we would need 10^9 threads to keep an exascale machine busy. To put this into context, a 1km resolution model will have approximately 5×10^8 mesh elements in the horizontal. The Met Office currently plans to target a 10km global model (5×10^6 mesh elements) for operational NWP in the 2018-2020 timeframe.

Whilst this naive analysis may suggest that such a global model has only enough data parallelism to exploit 10^6 or 10^7 concurrent threads, this is not as bad as it first appears. Neither operational forecasts nor research jobs occupy the whole machine and moreover, increased use of coupled models instead of atmosphere-only models can be expected to boost resource utilisation as can ensemble runs via task parallelism. However it is clear that in the future it will be necessary to find and exploit additional parallelism beyond horizontal mesh domain decomposition and concurrent runs of the atmosphere, ocean and IO models.

1.3 Design considerations for a dynamical core

The GungHo dynamical core is being designed from the outset with both numerical accuracy and computational scalability in mind and the primary role of the computational science workpackage is to ensure that the solutions being proposed by other workpackages in GungHo will scale efficiently to large numbers of threads. At the same time, we recognize that a software infrastructure for next generation weather and climate prediction must aspire to an operational lifetime of 25 years or more if it is to match the longevity of the current Unified Model. Therefore, other aspects – such as portable performance, ease of use, maintainability and future-proofing – are also important, and are addressed here.

Portable performance means the ability to obtain good performance on a range of different machine architectures. This should apply to the size of machine (from tablet to supercomputer), the model configuration (climate, NWP, coupled etc) and to different classes of machine (e.g. GPU oriented or many-core oriented). It is not known what type of architectures will dominate in 2018 and onwards but, as is the case today, architecture specific parallelisation optimisations are expected to play an important role in obtaining good performance. Therefore the GungHo software architecture should be flexible enough to allow different parallelisation options to be added.

Ease of use is primarily about making the development of new algorithms as simple as possible and ease of **maintenance** will hopefully follow from this simplicity. Clearly there is a trade-off between application specific performance optimisations and ease of use/maintainability. As will be shown in later sections, in the proposed GungHo software architecture this problem is addressed by separating out these different concerns into different software layers. **Future-proofing** in this context refers to being able to support changes in the dynamical core algorithm (such as a change of mesh or discretisation for example) without having to change all of the software.

© Joint Crown and NERC Copyright 2013

2 Programming Language

It is recommended that the GungHo dynamical core be written in Fortran 2003; that is, it may rely on any feature of the Fortran 2003 standard. This allows the support software to use some of the new Object Oriented features (function pointers within objects in particular) if required whilst keeping the performance benefits and familiarity of Fortran. Fortran 2003 also provides a standard way of calling C code, which is also recommended. Parallelisation is expected to be a mixture of MPI and OpenMP with the addition of OpenACC if GPUs are also to be targeted. However, there may emerge other programming models that these recommendations do not exclude. In particular Partitioned Global Address Space models (PGAS) such as Co-Array Fortran (CAF), which is part of the Fortran 2008 standard, are not excluded.

3 Quasi-uniform meshes

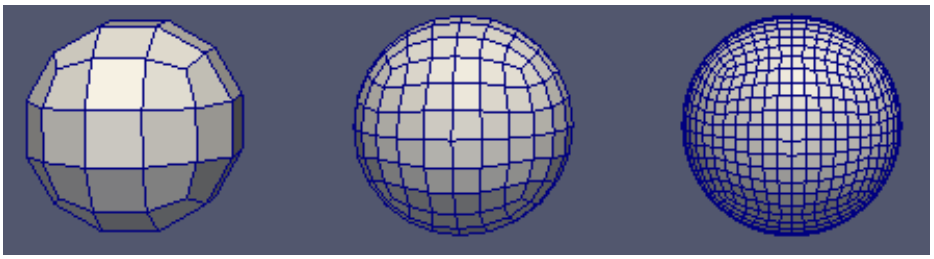

Figure 1: Cubed Sphere at increasing resolution

In the current Unified Model, a major factor limiting the scalability and accuracy of global models is the use of a regular latitude/longitude mesh, as the resolution increases dramatically towards the poles. GungHo's proposed solution to this problem is to use a quasi-uniform structured mesh. At the time of writing, the cubed-sphere is the most likely candidate – see Figure 1 for a view of the structure of a cubed sphere – but investigations are on-going. For more details please see the Phase one grids document (GungHo Team, 2013).

The cubed-sphere has a relatively simple structure and can be represented by six regular meshes each covering one side of a cube, or by other less regular meshes. It would therefore be possible to use arrays to directly store the connectivity of the mesh, allowing neighbours to be indexed in the familiar stencil style (`data(i+1,j+1)` *etc.*) of a structured mesh, with some swapping of indices at boundaries.

However, it is highly undesirable to specialise to a particular grid prematurely. At the time of writing, alternative grids are still under consideration in GungHo. Moreover, we should not presume that the best choice of grid now will still be the best choice over the lifetime of the GungHo dynamical core. Hence we must place a premium on flexibility, and we aim for an infrastructure that can support multiple grids. Therefore the computational science workpackage has pro-

© Joint Crown and NERC Copyright 2013

posed that a general irregular data structure be used to store the data on a mesh. This means that instead of neighbours being addressed directly they are addressed by a neighbours list *e.g.* `data(neighbour(i),i=1,Nneighbours)`.

Iterating over a list of neighbours allows an arbitrary number of neighbours for each mesh element which allows simple support for different topologies. Another difference between the explicit stencil structures and unstructured representations is that in the former case there are two dimensions in the index array for the horizontal whereas in the latter there is only one.

The price of flexibility is likely to be some loss in performance. The open question is how much.

There is a performance cost associated with having to look up the locations of neighbours. However, by using direct addressing in the vertical and traversing the vertical dimension in the innermost loop nest, the overhead of indirect addressing can be reduced to a negligible level. This is because there will only be one look-up for each column of data. Thus MacDonald et al. (2011) demonstrate that a set of kernels drawn from the FIM model execute at within 1% of the directly addressed speed when executed in a vertical direct, horizontal indirect framework.

By adopting a flexible ”unstructured” data model, we do not necessarily preclude grid-specific optimisations, some of which can be implemented ”under the hood” (for example, re-numbering for improved cache locality). But we do remove these concerns from view in the algorithm layer.

This is of course not the whole story. Fair comparisons between structured and unstructured approaches are hard to come by, because it is difficult to separate impact on performance of factors such as the order of the method; the choice of finite difference, finite volume or fine element approaches; the efficiency of the implementations; and so on. We will be in a position to report better estimates of the performance overheads once our investigations with prototypical kernels are further advanced.

Thus we are driven to a data model in which we use indirect addressing in the horizontal, and direct addressing in the vertical, with columns contiguous in memory. In contrast, the current Unified Model uses direct addressing in all three dimensions, and it is levels that are contiguous in memory. The change from column-contiguous (vertical innermost) to level-contiguous (horizontal innermost) will also have performance consequences, which depend on the operation being performed and the architecture on which it runs.

Some studies have shown that on architectures such as GPUs, horizontal innermost can provide improved performance. For a cache based memory model such as conventional CPUs, coding the vertical index as the innermost dimension is usually optimal for cache re-use because the array data in this dimension will be contiguous in memory. Thus, an inner loop over this dimension will exhibit data locality and allow the compiler to exploit the cache. However, for a non-cache based memory model such as on a GPU this arrangement may not necessarily be optimal. On current GPUs, it is best practice to hide memory latency by by oversubscribing the number of thread teams (commonly refererred to as *warps* for NVidia hardware and software) to physical cores. The low cost of switching warps whilst waiting for data amortises the cost of memory access. Consequently the

© Joint Crown and NERC Copyright 2013

number of concurrent threads is very high. Moreover, GPUs have a vector memory access feature (known as *coalesced memory access* in NVidia terminology) for an indiviual warp. To access data in this way, each thread in a warp requests data from a sequential element of a contiguous array in a single memory transaction. Naively, a data layout with the *horizontal* index innermost would achieve this. However, coalesced memory access could still be achieved with the vertical index innermost if a warp of threads cooperate to load a single element column into shared memory as a contiguous load. The warp then computes for the column (using a red-black ordering if required if write stencils overlap). In particular, a thread-wise data decomposition in the vertical is likely to be required for operations such as a tridiagonal solve so that data segment sizes are sufficiently reduced to fit into (warp) shared memory.

Furthermore, such considerations are not limited to GPUs. Commodity processors today possess cache-based memory architectures and vector-like, pipelined execution units capable of retiring multiple floating-point instructions per cycle. Although today's "vectors", typified by 128 or 256 bit SSE or AVX, are much smaller than in the era of vector computers, it is still the case that an application incapable of exploiting the Instruction Level Parallelism (ILP) that such units confer is likely to achieve disappointing performance. It is unclear whether the recent trend towards larger vector units in commodity processors will continue, because the returns for many applications – especially those of limited computational intensity – are diminishing. In a recent study of the NEMO ocean code, Pickles and Porter (2012) compared two alternative index orderings for 3-dimensional arrays; in the "level-last" ordering (which is used both by NEMO and the current UM), a layer is contiguous in memory; in the alternative, "level-first" ordering, a column is contiguous in memory, as will be the case in the proposed GungHo data model. The study found that although the level-first ordering is advantageous for some important operations (such as halo exchanges and the avoidance of redundant computations on land), and although most operations can be made to vectorize in either ordering, there remain some operations that are difficult to vectorize in the level-first ordering. The archetypical example is a tridiagonal solve in the vertical dimension, because of loop-carried dependencies that inhibit vectorization. In the level last ordering, the dependencies remain, but by operating on a whole layer at once, the vector units can be fed; however, this trick is cache-unfriendly in the level-first ordering. As it is likely that similar concerns will arise in the GungHo dynamical core, we are investigating alternative methods for solving tri-diagonal systems such as sub-structuring and cyclic reduction. These methods have in common the idea of transforming the problem into one that exposes more parallelism, albeit at the cost of more floating point operations; the hope is that as "flops become free", the increased parallelism will win.

Clearly, performance portability is a complex issue. One outstanding question is whether the software architecture can be designed so that data orderings could be changed in the future if necessary. The answer may be no, although small kernels (operating on a single cell-column) would be simpler to re-write especially if a domain specific language and auto-generated code are used. There are complex performance benefits and penalties whatever choice is made, and uncertainty

© Joint Crown and NERC Copyright 2013

in what hardware features will be available at exascale make all the consequences of this choice harder to foresee. However, our recommendation of vertical index innermost represents the best choice for achieving performance with the awareness that hardware specific performance code will be necessary to exploit whatever hardware features are available. This consideration informs the design of the GungHo software architecture.

4 Proposed Software Architecture

The general approach that is being proposed by the computational science workpackage is to formally separate the GungHo software into different layers each with their own responsibilities. These layers are each separated by an API.

These layers are the Driver, the Algorithm, the Parallelisation System (PSy), the Kernel and the Infrastructure. These are illustrated in Figure 2. The reason for this separation is that the different layers have different requirements and these requirements can then be treated in isolation. For example the PSy is responsible for the shared memory parallelisation of the model and the placement of communication calls. The layers above (Algorithm) and below (Kernel) know nothing about this.

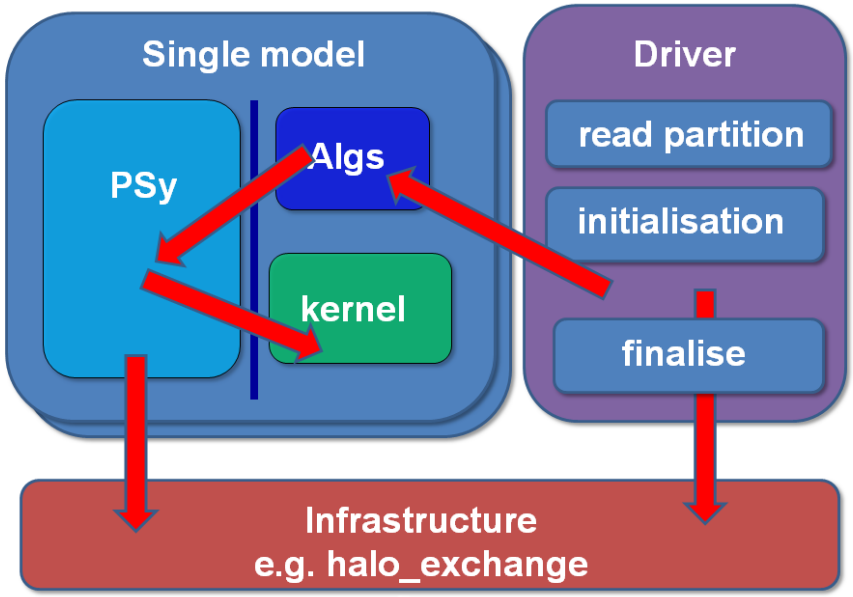

Figure 2: GungHo Component Diagram

The Driver knows about all models in a coupled model and their partitions; it calls the Algorithm layer in each model. The Algorithm layer is effectively the top level of GungHo. It is where the scientist defines their numerical algorithm in terms of components called kernels. The Algorithm layer calls the PSy layer which is responsible for shared memory parallelisation and intra-model

© Joint Crown and NERC Copyright 2013

communication. The PSy layer calls the scientific kernel routines. The kernel routines implement the scientific algorithm. All layers may make use of the infrastructure layer which provides services (such as halo operations and library functions). The following sections discuss each of the layers and their associated APIs in more detail.

Although the proposed software architecture is layered, some layers are depicted side by side in Figure 2. The reason for this, and the explanation for the vertical line between the Algorithm layer and Kernel layer on one side and the PSy on the other, is to emphasise the division of responsibilities. The Scientist will deal with the Algorithm and the Kernel, and the Performance Expert will deal with the PSy layer. The vertical line has been termed the "iron curtain" to underline the strong separation between code parallelisation and optimisation on the one hand and algorithm development on the other.

The kernel layer can be treated as part of a general toolkit in order to promote the re-use of functionality within GungHo and between GungHo and other models at the Met Office (including the Physics) to help avoid re-implementing the same functionality in different places. In a formal instantiation of this idea (see Glover, 2013), developers would be encouraged to use the toolkit functions wherever possible and would not be able to add new science into the system unless that science was added to the toolkit.

4.1 Driver layer

Strictly, the driver layer is external to the GungHo dynamical core. However, it calls the GungHo software, may manage some of the GungHo state (internal data) and is needed to run a model containing the GungHo dynamical core. Therefore, it warrants discussion in terms of its interaction with the GungHo software.

The driver layer is responsible for the **scheduling** of the individual models in a coupled model including the **dump** and **restart** frequencies. There may be different implementations of the driver layer depending on the particular configuration. For example, a stand-alone driver might look different to a driver for a fully coupled Earth System Model (ESM).

The driver is also responsible for holding any **state** that is passed into and out of the GungHo dynamical core, or exchanged with other models, and for ensuring that any required coupling involving the state takes place. The driver may know little about the content of the state. For example, it may simply pass an opaque state object, as is the case with the Model for Prediction Across Scales (MPAS) (MPA). However, these options need further investigation, and for now, we postpone the decision as to which layer(s) will be responsible for **allocating** and **initialising** the state variables.

The driver will control the **mapping** of the model onto the underlying **resources** (i.e. the number of MPI processes and threads and the number of executables). It is envisioned that the driver will may make use of existing coupled model infrastructure provided by third parties, for example some of the functionality of the Earth System Modelling Framework (ESMF) may be adopted.

© Joint Crown and NERC Copyright 2013

4.2 Driver/Algorithm interface

It is proposed that the driver calls the algorithm layer directly. For example, in the following code a module called `DynCore` contains at least 4 subroutines (there may be more that are not called), two of which are initialisation subroutines (so are typically called once at the start of a run), one of which performs a single timestep (so is typically called a number of times) and the last of which cleans up after the computation has completed (so is typically called once at the end of a run). In the example each of these subroutines takes in a state object (`DynCoreState`) which contains all state used by the DynCore module that is not declared locally within the module.

```
use DynCore, only : init1,init2,run,finalise
call init1(DynCoreState)
call init2(DynCoreState)
while .not. finished
  call run(DynCoreState)
end do
call finalise(DynCoreState)
```

This approach is consistent with best practice in model coupling. The details of coupling, such as how data is passed from one model to another, are beyond the scope of the GungHo project. However, given the importance of model coupling in ESM, it would be remiss of us to ignore how coupling fits within our proposed architecture.

4.3 Algorithm layer

The algorithm layer, named Algs in Figure 2, should be where the scientist specifies the GungHo algorithm(s). As such it should be kept as high level as possible to keep it simple to develop and manage. This layer will have a logically global data view *i.e.* should operate on whole fields. It is strongly recommended that data objects at the field level should be derived types. The same applies to other data structures, such as sparse matrices, that are visible at the algorithm layer.

This layer should primarily (and potentially solely) make use of a set of calls to existing building block code and control code to implement the logic of the algorithm. These building blocks will often, but not exclusively, be defined by user-defined kernels that specify an operation to be performed, typically column-wise, on fields. The exception is common or performance critical operations that may be implemented as library functions. Calls may be in a hierarchy, for example a call to a helmholtz solver may contain a number of smaller "building block" calls.

There will be no explicit parallel communication calls (for example halo updates) in the Algorithm layer: this will be handled exclusively by the PSy layer. However calls to communicate data via a coupler (`put/get` calls) *will* occur at this layer.

© Joint Crown and NERC Copyright 2013

As the example in the previous section implies, it is recommended that the Algorithm layer is implemented as a module with a number of public subroutines. An example is given below.

```
module DynCore
implicit none
private
public :: init1,init2,run,finalise,dump
contains
  subroutine init1(DynCoreState)
  !...
  end subroutine init1
  subroutine init2(DynCoreState)
  !...
  end subroutine init2
  subroutine run(DynCoreState)
  !...
  end subroutine run
  subroutine finalise(DynCoreState)
  !...
  end subroutine finalise
  subroutine dump(DynCoreState)
  !...
  end subroutine dump
end module DynCore
```

This example corresponds to the driver example in the previous section. It has one additional `dump` routine which is meant to demonstrate a natural way in which a restart-dump routine could be implemented. This layer will be implemented in Fortran but could be replaced with a domain specific language in the future.

4.4 Algorithm to PSy interface

The Algorithm/PSy interface is still under discussion and requires further study. The options being evaluated are outlined below.

1. The Algorithm layer provides functions to the PSy layer which are executed or computed. In this case the Algorithm layer passes the required function (implemented in the Kernel layer) as an argument to a generic compute function. In the example below the compute function is called `compute` and it passes the `sub` subroutine along with the required subroutine arguments. The PSy layer would then need to implement a generic compute routine which

© Joint Crown and NERC Copyright 2013

performed the appropriate actions depending on which subroutine was passed. Advantages of supporting a single generic compute function through which all algorithm functions are passed are that a single operation in the PSy layer (such as partitioning for example) could operate on all specific functions and that this solution also fits well with code generation tools.

```
use kernel, only : sub
use PSy, only : compute
call compute(sub,data1,data2,...)
```

2. The Algorithm layer calls a PSy layer routine directly which in-turn calls a Kernel layer routine. In the example below the Algorithm layer does not know about the Kernel layer and simply calls functionality that has been made available by the PSy layer.

```
use PSy, only : sub
call sub(data1,data2,...)
```

In both of the above cases there is a potentially serious performance issue which needs to be addressed. The PSy layer, as discussed in the next section, is responsible for implementing any shared memory parallelisation and placement of intra-model communication (such as halo swaps). If each routine is called in isolation then it is only possible to parallelise and optimise that routine in isolation, however there may be significant performance benefits in parallelising a number of routines together.

The idea of the Algorithm layer calling a function in the PSy layer which in turn calls a set of functions has been rejected as it can lead to "algorithm creep" from the Algorithm layer into the PSy layer. As an example, consider the helmholtz function below which is implemented as a set of lower level operators. Each of these lower level operators could use either of the options detailed above. However, if we want to optimise all functions together (e.g. block them for cache use) then we need to modify the set of lower level operators as a group. This would require moving the helmholtz functionality into the PSy layer as we are not allowed to optimise code in the Algorithm layer. The implementation of the helmholtz function would then appear in the PSy layer leading to algorithm creep. It could be the case that for performance purposes a large part of the algorithm would end up being called from the PSy layer.

The proposed solution is to revisit the idea of passing functions as arguments. Again there are two potential solutions here. The generic solution would extend the PSy `compute` functionality to support multiple functions. The example below illustrates this solution where we have simply extended the generic compute function to allow multiple kernel functions to be passed.

```
subroutine helmholtz(...)
  use PSy, only : compute
  call compute(   &
```

© Joint Crown and NERC Copyright 2013

```
    HodgeI,...,    &
    Ddual1,...,    &
    HodgeH,...,    &
    Mult,...,      &
    Dprimal2,...,  &
    Sub,...    )
end subroutine helmholtz
```

A function specific solution (illustrated below) would be the same as the generic solution but have a specific name for the function in the PSy layer. The arguments for and against each of the approaches are the same as for the single function call except that it is arguably more complex to generically support multiple calls than single calls.

```
subroutine helmholtz(...)
  use PSy, only : compute_helmholtz
  call compute_helmholtz(    &
    HodgeI,...,    &
    Ddual1,...,    &
    HodgeH,...,    &
    Mult,...,      &
    Dprimal2,...,  &
    Sub,...    )
end subroutine helmholtz
```

Both interface options will support intent or data access descriptors. For example, specifying if the data passed is read only, write only or read and write by the function it is passed to. The Fortran `INTENT` attributes are insufficient as they only specify access for local scope and is not possible to programatically reason about data access patterns. A good example of this is automatic determination of whether data values in halos need to be updated. Data objects such as Derived Data Types (DDTs), will be passed from the algorthim layer to the Psy layer and whether the data intent is passed as an argument of the interface or a data member of the data object is not design critical.

The apparent differences between these two approaches are not profound. The generic `compute` routine needs to exist for development purposes: a science (numerics or physics) programmer needs to be able to write and test code without the intervention of a computational scientist to modify the PSy layer. Once code is correct, the PSy layer needs to make it fast. This might well be by manually replacing the `compute` call with a hand-optimised version, or it might involve an automated system producing the optimised version. The optimised version might have inlined the kernel code or have hand-reordering of halo operations such that changing the set of operations passed into the PSy layer invalidates that code. Accordingly, the hand-coded PSy layer routine should

© Joint Crown and NERC Copyright 2013

check the arguments passed (even if it doesn't actually call them) and cause an error if they have changed. A science programmer who wishes to make changes to the algorithm will switch back to calling `compute` for development purposes, and then re-engage with the manual or automatic PSy layer once those changes are correct.

4.5 PSy layer

The Parallelisation System (PSy) layer is the domain of the performance expert. This is where shared memory parallelisation can take place (the MPI partitioning is performed at the driver layer through calls to the infrastructure), single core optimisations such as blocking and placement of communication calls, such as halo operations.

It is expected that machine specific solutions will be required to obtain good performance on different machine types (many core vs GPU based for example), however such machine specific solutions may be limited to certain functions allowing many functions to have a "vanilla" parallelisation solution.

The PSy layer is invoked by the Algorithm layer via the interface described in the previous section. In the case of multiple functions being passed, the PSy layer will be able to decide the most appropriate ordering of these functions, including whether they are to be computed concurrently. It is the responsibility of the PSy to access the raw data from the DDTs passed through the interface.

The PSy layer calls routines from the Kernel layer. The implementation will also be able to call routines from the infrastructure layer. The Kernel functions are made available as a module and the PSy layer simply uses this module and calls the appropriate function(s).

There is likely to be a vanilla version of the PSy layer for a supported algorithm which performs little or no optimisations. This can be useful for debugging purposes. The vanilla version can be specialised to create optimised versions.

The PSy manages the placement of communication calls, such as halo operations. Halo operations are traditionally manually placed in the code in the appropriate places. This can be done in the PSy layer and an API will be provided in the infrastructure layer. It is expected that different architectures will benefit from different placements of communication calls. For example, some platforms benefit from early posting of receives and some platforms benefit from the overlapping of computation and communication. An alternative is for the system to automatically apply halo operations when required. This works by providing information about which kernel routines require halo data and which kernel routines update halo data then triggering an appropriate halo operation either in an eager or a lazy way.

4.6 PSy to Kernel interface

Whatever the implemetation of the Algorithm to Psy interface, the PSy layer calls the Kernel routines directly, passing data in and out by argument with field information being passed as raw arrays.

© Joint Crown and NERC Copyright 2013

Two solutions are being evaluated for the PSy to Kernel interface. It may be that some combination of the two is recommended. These are discussed in the following section.

An example using a malleable kernel routine and an explicit call between the Algorithm and PSy layer is presented below. In this example the routine `HodgeI` is called from the PSy layer. The kernel code happens to have the same name so is renamed when it is used. The data from the PSy layer is passed as a derived data type so the actual data is extracted. The kernel code is then called in parallel using openMP and also each thread is given a number of segments to improve cache use.

```
subroutine HodgeI(a,b)
use kernels, only : HodgeI=>HodgeI_kern
aData=>a%data
bData=>b%data
!$OMP DO PRIVATE(lo,hi)
do i=1,nsegs*nthreads
  lo=...
  hi=...
  call HodgeI_kern(lo,hi,aData,bData)
end do
!$OMP END DO
end subroutine HodgeI
```

4.7 Kernel layer

The Kernel layer is where the building blocks of an algorithm are implemented. This layer is expected to be written by an algorithm developer with input from a performance expert as this layer should be written for ease of maintenance but also such that the compiler can optimise the code to run efficiently.

To help the compiler optimise, explicit array dimensions should be provided where possible. In certain circumstances, where performance of a routine is critical it may be that compiler directives might need to be added to the kernel code to aid optimisation (such as unrolling or vectorisation). In more extreme cases there may need to be more than one implementation of a particular kernel which is optimised for a particular class of architecture, but this should be avoided if possible.

The Kernel routines will only be called from the PSy layer and will be implemented as subroutines within a module with data being passed into and out-of the routine by arguments. The data will be passed in its raw form to aid compiler optimisations.

Two potential kernel implementations have been proposed. These are not mutually exclusive and it may be that some mixture of the two is used. In the first implementation the parallelisable dimensions are made malleable, so that the parallelisable dimensions can be called with different

© Joint Crown and NERC Copyright 2013

numbers of work units. This approach is flexible in that the same Kernel code can be used for smaller or larger numbers of work units per call depending on what the architecture requires. For example, one would expect a GPU machine to require a large number of small work-units and a many-core machine to require fewer larger work units.

In the second proposed solution the loop structures are not contained in the kernel code and any looping is performed by the PSy layer. One exception to this rule is the loop over levels which may be explicitly written, particularly if it is not possible to parallelise the loop.

Another ongoing discussion is to do with the recommended size of the kernels, although there is nothing in the architecture design that dictates kernel size. Smaller kernels (in terms of amount of computation) are more suitable for architectures based on GPUs but larger kernels are more suitable for multi-core processors. It may be that smaller kernels can be efficiently combined to create larger kernels which would tend to favour smaller ones. Ultimately it will be the unit of work defined by the algorithm developer that will dictate the size of a kernel. In the case of Physics routines it may be that these will be integrated in an evolutionary way (re-using existing code and starting with large kernels) rather than a revolutionary one (a full code re-write).

Finally, the Kernel layer will be managed as a toolkit of functions which can only be updated by agreement with the toolkit owner. Combined with a rule that disallows code, other than subroutine calls and control code, at the algorithm layer, this approach would promote the idea of re-using existing functionality (potentially within across different models, not just the GungHo dynamical core) as code developers would need to apply to add new functionality to the kernel layer and this could be checked with the existing toolkit for duplication.

4.8 Infrastructure layer

The infrastructure layer provides generic functionality to all of the other layers. It is actually a grouping of different types of functionality including **parallelisation**, **communication**, **libraries**, **logging** and **calendar** management.

Parallelisation support will include functionality to support the partitioning of a mesh and the mapping of (MPI) processes onto the partitions. It is expected that ESMF or some similar software solution will be used to support this functionality but it is proposed that this functionality is not provided directly but rather through the infrastructure layer so that we only expose the appropriate functionality.

Communication support is related to the partitioning support as halo operations are derived from the specification of a partitioned mesh in ESMF. Again it is proposed that this functionality is not provided directly but rather through the infrastructure layer so that we expose the appropriate functionality. The infrastructure will also provide the coupling put/get functionality (probably implemented by OASIS3) but that is out of scope for GungHo.

It makes sense to use standard optimised libraries (such as the BLAS or FFTs) where possible

© Joint Crown and NERC Copyright 2013

and the infrastructure will provide a subset of these to the user.

Finally, logging support is a useful way to manage the output of information, warnings and errors from code and calendar support helps manage the timestepping of models. ESMF provide infrastructure for both of these and the infrastructure will provide the appropriate aspects of these to the user.

5 Data model

A data model is an abstract description of the data consumed and generated by a program. The fundamental form of data employed in simulation software is the field. That is, a function of space and, potentially, time. A data model for fields therefore sets out to express field values and their relationship to space and time. In simulation software, the field is represented by a finite set of degrees of freedom associated with a mesh and potentially changing through time. Simulation software must therefore have a data model which encompasses meshes and fields, and gives the relationship between them.

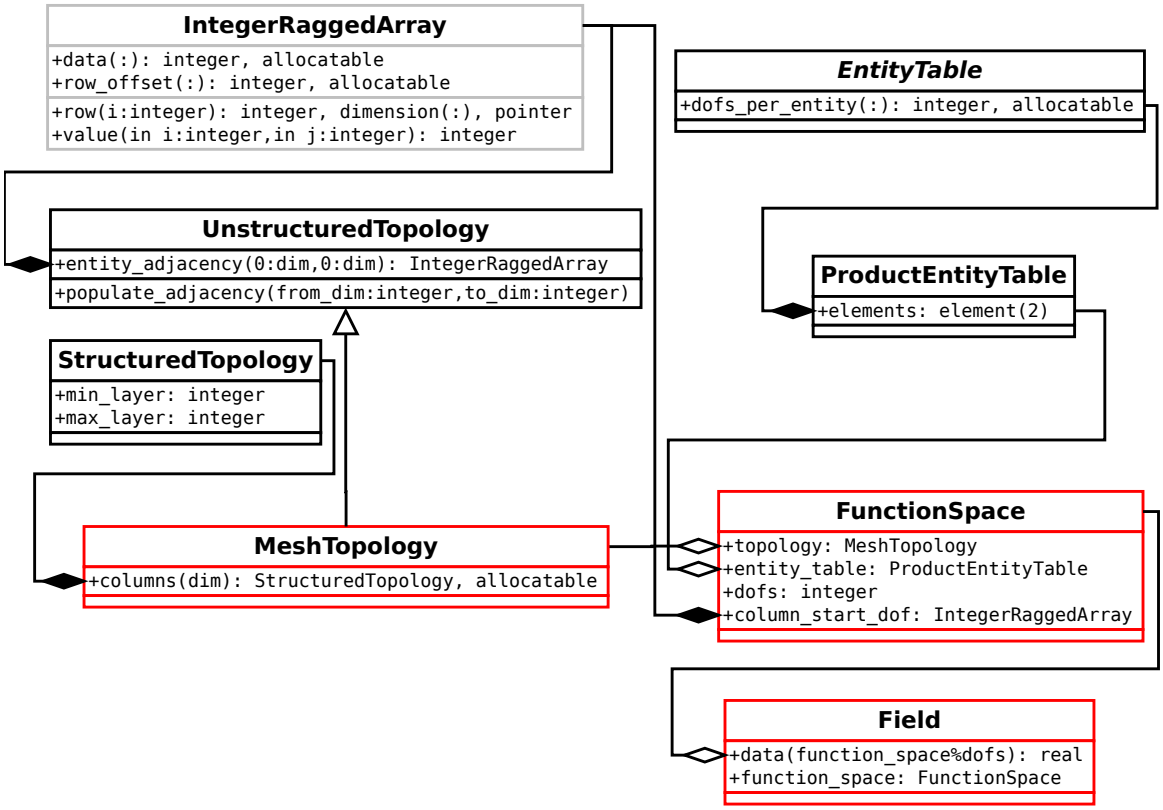

Figure 3: UML diagram showing the key data objects in the recommended data model. The most important components are the `MeshToplogy`, `FunctionSpace`, and Field types.

© Joint Crown and NERC Copyright 2013

5.1 Data models for Gung Ho meshes

Rectangular structured mesh models have meshes whose structure maps naturally to the structure of multidimensional arrays. Since this mapping is so natural, and frequently only implicitly represented in the model code, it is often possible to forget that the data model exists at all. For an unstructured mesh an explicit data model is necessary.

The data model will necessarily pervade the model code. This has important implications for choices about generality in data model. As the existing UM infrastructure demonstrates, the expected life of the code base as a whole far exceeds the life of the dynamical core. It would therefore be particularly unfortunate for the data model to incorporate assumptions about the functioning of the dynamical core which might not be true of the next generation. Furthermore, it will be argued here that it is not necessary that it do so. On the other hand, there are some features of atmospheric models which appear so fundamental that they would appear in any model and so may usefully be assumed by the data model.

5.1.1 Horizontally unstructured columnar meshes

One assumption which seems particularly safe is that the mesh will be organised in columns. Section 3 examines the range of meshes which might be employed in Gung Ho and in particular, recommends that indirect addressing be adopted in the horizontal, with direct addressing employed in the vertical. For this to be an efficient operation, the data model will have to reflect this choice inherently by ensuring vertically aligned degrees of freedom are arranged contiguously in memory.

5.1.2 Choice of discrete function space

Models approximate functions by storing the value of a finite number of Degrees of Freedom (DoF). The relationship between these DoF and the mesh is determined by the discretisation adopted for that field. For example, the C grid associates a single flux DoF with every mesh facet, while the A grid associates one velocity DoF per dimension with each vertex. To take a slightly more unusual case, the BDFM element (introduced for weather applications in Cotter and Shipton, 2012) associates two flux DoF with each triangle edge and three flux DoF with the triangle interior. Spectral element schemes take this much further with very large numbers of DoF associated with each topological entity.

It is therefore recommended to allow for a wide variety of function spaces to be represented, including those which associate more than one DoF with a given topological entity, and those in which a single field has DoF associated with more than one kind of topological object.

© Joint Crown and NERC Copyright 2013

5.2 Key data objects

The fundamental objects in the data model are shown in figure 3. The field object contains the DoF values which describe the field value. The function space describes the relationship between the DoF and the mesh: with which topological entities the DoF are associated, how many DoF with each and so on. In a finite element model, the basis functions are properties of the function space. For suitable finite difference models, lifting operators and differential operators may be able to be described at function space level.

Entity	Dimension	Codimension
vertex	0	D
edge	1	D-1
face	2	D-2
facet	D-1	1
cell	D	0

Table 1: Names for topological entities in a D-dimensional mesh (from Logg, 2009). Observe that cell and facet are relative terms: in a 2D mesh the cells are two dimensional and the facets are edges.

5.3 Mesh topology

The mesh describes the discretisation of the domain into cells. The mesh is the basis for the representation of the geometry of the domain and all of the system state fields on it. An excellent abstraction of the concept of a mesh topology is given in Logg (2009). This abstraction will be followed here, with suitable extensions to enable the explicit representation of columnar meshes.

A mesh may be considered as a purely topological object composed of entities. In Logg's taxonomy, these have the names given in table 1. The advantage of a generalised concept of entities is that it facilitates treating meshes of different dimension in similar ways. This will prove useful in treating a columnar mesh as a 2d mesh of columns.

Logg's topology model is illustrated in figure 4. Note that we can uniquely reference any entity using a dimension, id pair. The initial information required is, as usual, the identity of each entity in terms of its incident vertices (0-entities). Logg provides efficient algorithms to derive any other incidence relationship (the facets incident to each cell, for example).

5.3.1 Extending the mesh topology to incorporate columns

The topology model described here is fundamentally indirectly addressed. However as discussed in section 3, it is recommended that Gung Ho adopt a mesh which is horizontally unstructured and indirectly addressed, but vertically structured and directly addressed.

This can be achieved in this model by creating a 2D topology of columns, each of which is a contiguous list of entities aligned in the vertical. This column topology is actually the Cartesian

© Joint Crown and NERC Copyright 2013

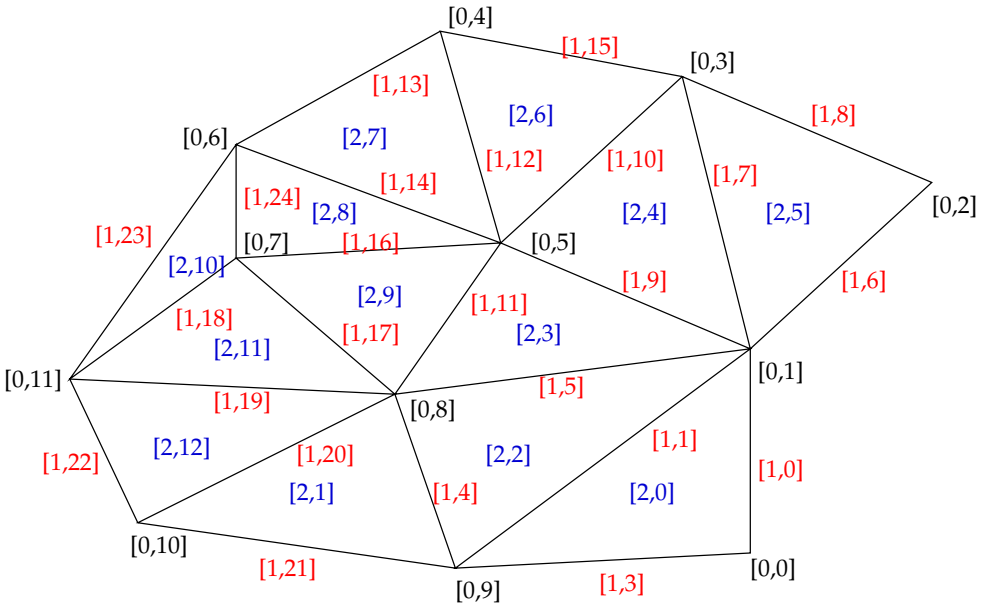

Figure 4: Example 2D topology, T. 0-entities are numbered in black, 1-entities in red and 2-entitqies in blue.

product of the original 2D topology and a new 1D topology (if the possibility is allowed of variable layer counts throughout the domain, this will be a 1D topology for each entity in the 2D topology).

Entities in the column topology have the sum of the dimensions of the corresponding 1D and 2D entities. For each entity of the 2D topology, with entity dimension given by d, the corresponding column will therefore have entities of dimension d and $d + 1$. For instance, each 2-entity of the column mesh is a column made up of 3-entities (cells of the 3D mesh) and 2-entities (the horizontal facets of the 3D mesh). The 1-entities (edges) of the 2D mesh are the vertical facets and horizontal edges of the 3D mesh. Finally, the 0-entities (vertices) of the 2D mesh contain the vertical edges and the vertices of the 3D mesh.

The relationship of the 3D entities to the 2D mesh is important for the construction of function spaces and discretisations: for example a horizontal face is not equivalent to a vertical face. This distinction may be established by labelling entities in the 3D mesh by the corresponding pair of entities from the 2D and 1D meshes respectively. For instance vertical faces are created by the Cartesian product of edges of the 2D mesh and edges of the vertical mesh, and are therefore $(1, 1)$-entities. Conversely, horizontal faces are the product of faces in the 2D mesh and vertices in the 1D mesh and are consequently $(2, 0)$-entities.

5.3.2 Implementing the topology

The two-dimensional unstructured mesh is purely a graph of incident topological entities. The incidence relationships between these entities can conveniently be represented by an integer ragged array of adjacencies. In the most general case, one such array is required for each pair of dimensions in the range $0 \ldots 2$, and the `UnstructuredTopology` in figure 3 implements this. For a

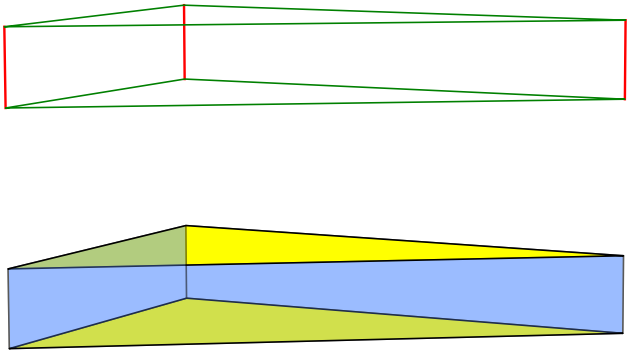

Figure 5: Two images of a triangle extruded to form a wedge element. In the upper figure, the $(1,0)$ edges are shown in green while the $(0,1)$ edges are shown in red. The lower figure illustrates the faces of the wedge, the $(2,0)$ faces in yellow and the $(1,1)$ faces in blue.

given model and set of discretisations, some subset of these will be required. It is only necessary to actually generate those which are used by a particular model. The `populate_adjacency` method is provided for this purpose.

The vertical topology is very simple indeed, since there is only one topological mesh in one dimension for any given layer count. The `StructuredTopology` object therefore only contains a minimum and a maximum layer. Supporting varying layer counts in different parts of the domain (for instance to support z-layer models) could be implemented using a mask in a similar manner to a fully structured mesh. The full 3D `MeshTopology` object simply combines an `UnstructuredTopology` with a `StructuredTopology` by inheriting from the former and including an instance of the latter as a field (multiple inheritance is not supported in Fortran 2003). In numbering layers there is a fencepost issue, so a `StructuredTopology` with n-layers will have n edges and $n+1$ vertices. The convention is adopted that the edge shares an index with the vertex below it, so $[1,a]$ is the edge above $[0,a]$.

This is a sketch implementation and in particular omits details concerned with halos for parallel communication, however these do not present difficulties unique to Gung Ho.

5.3.3 The mesh topology provides the iteration spaces

One point which may be less obvious in code with no explicit separation between fields and topology is that the spatial loops in the code are actually loops over topological entities. For instance, the code may loop over all of the cells to calculate scalars, or it may loop over all the facets in order to compute fluxes. In a given discretisation, there may be a one-to-one correspondence between flux DoF and facets, but it is the adjacency information which is required to access the inputs to the

© Joint Crown and NERC Copyright 2013

calculation, and this information is a property of the topology.

5.4 Function spaces

From the perspective of the data model, the role of a function space is to associate DoF with topological entities. For example, which DoF are incident to a particular cell or facet? In other words, it is the function space which implements the addressing (direct and/or indirect) which enables local discretisation operators to be applied to fields.

Function spaces have both local and global components. The local level describes how many DoF are associated with particular topological entities, while the global level contains the map(s) from entities to DoF numbers. The local information depends on the discretisations employed, but not on the mesh on which a simulation is run. Conversely, the global function space depends on the particular mesh topology and therefore varies between simulations conducted on different meshes (for instance, simulations at different resolutions).

5.4.1 Local function space information.

At local level, the function space needs to record the number of DoF associated with each class of topological entity. For certain discretisation types, it may prove convenient to store further information at this level (basis functions for finite element models, or lifting operators for certain types of finite difference discretisation), however this additional information is orthogonal to, and beyond the scope of, the core data model question.

Entity dimension	0	1	2
DoF	0	1	0

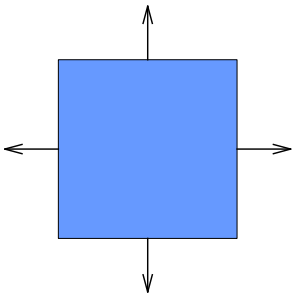

Figure 6: Entity table for 2D C-grid velocity. There is one degree of freedom associated with each edge.

Entity tables The relationship between Entities and DoF can be represented using entity tables. Consider first some two dimensional discretisations which might be employed for velocity elements. The current Met Office choice is the C grid (analogous to the Raviart-Thomas 0 finite element), which associates one degree of freedom with each facet. Figure 6 shows the entity table for this discretisation. An alternative under consideration in the Gung Ho project is the first Brezzi-Douglas-Fortin-Marini element (BDFM-1). The entity table for this element is shown in figure 7. More gener-

© Joint Crown and NERC Copyright 2013

Entity dimension	0	1	2
DoF	0	2	3

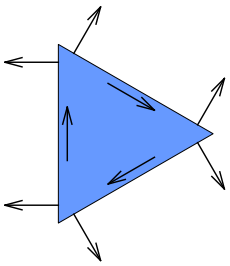

Figure 7: Entity table for 2D BDFM1 velocity. There are two degrees of freedom associated with each facet and 3 with each cell.

	0	1
0	0	0
1	0	1
2	0	0

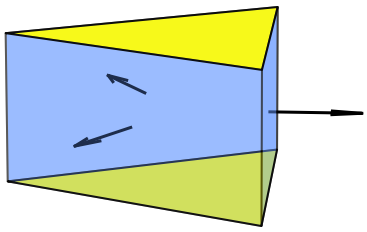

Figure 8: Entity table for horizontal C-grid velocity extruded in the vertical using a piecewise constant distcretisation. There is one degree of freedom per $(1,1)$-entity.

ally, the degree of freedom placement for *any* discretisation can be expressed in this form, and this entity table may be combined with the mesh topology to produce a global numbering of degrees of freedom.

Cartesian product entity tables The columnar meshes envisaged here are designed to work with columnar discretisations, which the elements are prisms with polygonal horizontal cross-section. A horizontal discretisation therefore needs to be combined with a vertical discretisation to produce the discrete fields over the corresponding wedge elements. A common mechanism for achieving this is to take the product of a horizontal discretisation with a vertical one. For instance, the natural extension of the C-grid horizontal velocity to a layered mesh is to hold the velocity piecewise constant in the vertical direction. The entity table of the 2D C-grid is $[0,1,0]$ while that of a piecewise constant function in 1D is $[0,1]$ (one degree of freedom associated with each 1D cell). The entity table for the combined element can be found by taking the outer product of these tables. Figure 8 shows the entity table for this discretisation. As another illustration, assume we wish to represent a scalar field using a piecewise quadratic horizontal discretisation (entity table $[1,1,0]$) and a piecewise cubic vertical discretisation (entity table $[1,2]$). This table is shown in figure 9.

© Joint Crown and NERC Copyright 2013

	0	1
0	1	2
1	1	2
2	0	0

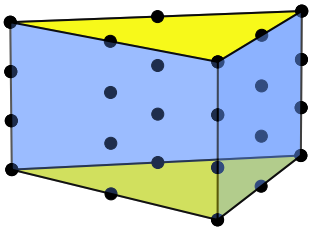

Figure 9: Entity table for a scalar field represented by a quadratic horizontal, cubic vertical discretisation.

5.4.2 Global function space information

The global component of the function space relates the local degrees of freedom with the mesh employed for a particular simulation. Unlike the local information, which is invariant for all simulations conducted with a given set of discretisations, the global function space information changes when the mesh changes. This makes the global function space information a run-time variable (although it might be cached for repeated executions on the same mesh).

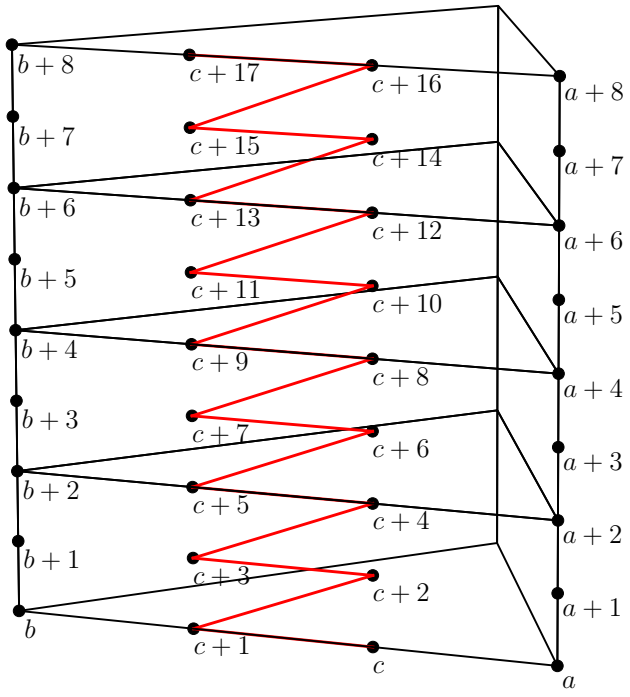

Figure 10: DoF numbering scheme for a scalar field which is cubic in the horizontal direction and quadratic in the vertical. Numbering is show for the DoF associated with the columns above two horizontal vertices (a and b), and for the column above c, the horizontal edge joining a and b. The red line joins adjacent DoF in the numbering of the column above c.

Associating DoF with topological entities The benefits of the vertical structure of the mesh are realised by ensuring that DoF which are vertically adjacent have adjacent indices and are therefore adjacent in memory. Figure 10 illustrates this arrangement for a scalar function space which is

© Joint Crown and NERC Copyright 2013

discretised with cubic polynomials in the horizontal and quadratic polynomials in the vertical. The entity table for this case is shown in table 2. Where a horizontal entity has more than one associated

	0	1
0	1	1
1	2	2
2	1	1

Table 2: The entity table for the function space whose numbering is demonstrated in figure 10. The DoF associated with the $(2,0)$ faces and $(2,1)$ cells are not shown in figure 10.

DoF (for example the horizontal edges in figure 10), these DoF are numbered adjacently in a zigzag pattern. It is assumed that these DoF are likely to always be accessed together, so this is numbering ensures optimal spatial locality.

5.4.3 Implementing function spaces

A possible function space implementation is sketched in figure 3. A mesh topology is associated with the product entity table associated with the discretisation to be represented. The numbering presented above is characterised completely by the entity table and the initial DoF associated with each 2D entity. This is stored in a 2D ragged array so that if `F` is a `FunctionSpace`, `F%column_start_dof(d,n)` is the initial DoF associated with 2D entity $[d,n]$.

5.5 Field objects

Field objects are in some sense the simplest objects in the data model. They simply associate a list of DoF values with a function space. It may be useful to allow fields to carry a data rank in addition to that in the function space. This would allow tensor quantities, and vector quantities in some function spaces to be represented without excessive duplication of function space information.

5.6 Geometry

Geometry has not yet been mentioned in this document. The mesh objects described are purely topological and have no concept of geometry. The construction of function spaces and fields has no dependency on the geometric location of the mesh entities, only on their connectivity. Geometry is, in fact, simply a discretised vector field over an appropriate function space associated with the mesh topology. Often, the geometry DoF are associated with the vertices (the $(0,0)$-entities) and the geometry is assumed to vary linearly, however the geometry may be represented by higher degree polynomials or in polar coordinates. The data model is agnostic to these decisions.

© Joint Crown and NERC Copyright 2013

5.7 Addressing

Strictly, addressing is a part of the execution model, however it is so intimately intertwined with the data model that it is worth mentioning here.

The typical pattern of access to a function space is to visit all of the topological entities of one class, and conduct a calculation over the DoF adjacent to that entity in a given sense. For example:

- a finite element integral over a cell (a $(2,1)$ entity) will access the DoF in the adjacent faces, edges and vertices;
- a finite element facet integral will access data from the adjacent cells and all the entities adjacent to those cells;
- a finite difference gradient operator at a facet will access the adjacent cell data (this is actually just a special case of the previous example).

Other typical access patterns are variations of these. For example, column "physics" operates on a horizontal entity, possibly some adjacent entities, and all of the DoF in the columns above them. An addressing algorithm for a finite element integral over a cell is described here, the corresponding variants for other access patterns follow in a similar manner.

The outer loop occurs over columns. Assume we have reached column $[2,c]$. Assume without loss of generality that the cell is triangular, and that we are accessing the field, f, whose entity table is given in table 2. Define

$$\mathrm{DoF}(d,n) = \texttt{f\%column_start_dof}\,(d,n) \tag{1}$$

and

$$v_i = \mathrm{DoF}(0, \texttt{f\%function_space\%entity_adjacency}\,(2,0)\,\texttt{\%value}\,(c,i)\,) \tag{2}$$

$$e_i = \mathrm{DoF}(1, \texttt{f\%function_space\%entity_adjacency}\,(2,1)\,\texttt{\%value}\,(c,i)\,) \tag{3}$$

$$c_0 = \mathrm{DoF}(2,c). \tag{4}$$

The DoF which will be accessed by the integral over the bottom cell will be:

$$\begin{aligned}\mathrm{DoFmap}(2,c) = [\,& v_0, \quad v_0+1, \quad v_0+2,\\ & v_1, \quad v_1+1, \quad v_1+2,\\ & v_2, \quad v_2+1, \quad v_2+2,\\ & e_0, \quad e_0+1, \quad e_0+2, \quad e_0+3, \quad e_0+4, \quad e_0+5,\\ & e_1, \quad e_1+1, \quad e_1+2, \quad e_1+3, \quad e_1+4, \quad e_1+5,\\ & e_2, \quad e_2+1, \quad e_2+2, \quad e_2+3, \quad e_2+4, \quad e_2+5,\\ & c_0, \quad c_0+1, \quad c_0+2\,].\end{aligned} \tag{5}$$

© Joint Crown and NERC Copyright 2013

This calculation, which occurs once per column, encompasses all of the indirect addressing operations. As the iteration progresses up the column, the DoF accessed by each successive cell integral are calculated using successive offsets from the DoF for the bottom cell. The offset between cells adjacent in the vertical for DoF associated with horizontal entities of each dimension is given by the matrix product of the entity table with the vector $[1, 1]$. This vector represents the single vertex and single edge difference between adjacent layers in the 1D structured topology. In this example,

$$\mathrm{inc} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 2 \end{bmatrix} \tag{6}$$

In this example, the offset array would be:

$$\begin{aligned} \mathrm{DoFinc}(2, c) = [\, & \mathrm{inc}(0), \quad \mathrm{inc}(0), \quad \mathrm{inc}(0), \\ & \mathrm{inc}(0), \quad \mathrm{inc}(0), \quad \mathrm{inc}(0), \\ & \mathrm{inc}(0), \quad \mathrm{inc}(0), \quad \mathrm{inc}(0), \\ & \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \\ & \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \\ & \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \quad \mathrm{inc}(1), \\ & \mathrm{inc}(2), \quad \mathrm{inc}(2), \quad \mathrm{inc}(2)\,] \end{aligned} \tag{7}$$

$$\begin{aligned} = [\, & 2, \quad 2, \quad 2, \\ & 2, \quad 2, \quad 2, \\ & 2, \quad 2, \quad 2, \\ & 4, \quad 4, \quad 4, \quad 4, \quad 4, \quad 4, \\ & 4, \quad 4, \quad 4, \quad 4, \quad 4, \quad 4, \\ & 4, \quad 4, \quad 4, \quad 4, \quad 4, \quad 4, \\ & 2, \quad 2, \quad 2\,]. \end{aligned}$$

5.8 Parallelism

There are three levels of parallelism which potentially interact with the data model: distributed memory parallelism implemented via MPI (or potentially a PGAS successor) and independent threads implemented by OpenMP (or a successor). Vectorisation is a separate issue and is neither facilitated nor excluded by this data model.

© Joint Crown and NERC Copyright 2013

5.8.1 Distributed memory parallelism

The data model interacts with distributed memory parallelism through the definition of halos which will be used to exchange data between adjacent mesh partitions. In this data model, it is the mesh topology which encodes the structure of the problem. It is envisaged that an external partitioning algorithm (regular or unstructured) will determine the ownership by processes of the vertices of the 2D mesh. It is then possible to determine the ownership of the 2D edges and cells by a voting algorithm. For example, a topological entity may be owned by the owner of the incident vertex with the lowest MPI rank. The ownership of all entities in the 3D mesh is the ownership of the corresponding 2D entity.

It is possible to determine those topological entities which will be involved in halo exchanges using the mesh adjacency and halo ownership information. The halo exchanges for any function space are then determined by inheritance from the mesh: a DoF will participate in a halo exchange if the topological entity with which it is associated would participate in that exchange. It is also possible to use the topological information to facilitate overlapping communication with computation during halo exchanges. A suitable algorithm is presented in Mudalige et al. (2012). Entities in that paper should be read as columns in this data model.

5.8.2 Colouring and threading

In employing heavyweight threads on conventional CPU architectures (as opposed to GPU threads which are more akin in this context to vector lanes), the key considerations are preventing data access conflicts between threads, and ensuring that memory accesses are as local as possible. It is the former which interacts most directly with the data model. The mesh topology inside a process can be divided into working sets each composed of one or more columns. The optimal working set size is likely to be determined by the amount of data which will fit into a relevant cache level. Using the mesh topology and the entity tables for fields written to by an operation, it will be possible to colour the working sets to prevent threads causing write conflicts by working on adjacent sets simultaneously. The available threads will then cooperate to complete the working sets of each colour in turn. This strategy is presented in more detail in Mudalige et al. (2012).

6 Conclusion

We have presented the recommendations of the computational science work package as they were understood at the conclusion of Phase 1 of the GungHo project.

Our recommendations for proposed software architecture and data model are designed to satisfy (sometimes conflicting) requirements for performance, scalability, and portability to emerging architectures; strong modularity, and maintanability; coupling to other models; and the flexibility to support at least multiple grids (if not multiple discretizations).

© Joint Crown and NERC Copyright 2013

There are numerous open questions concerning the details of the proposed architecture. We are in the process of answering these questions through the development of prototypes, in which we are exploring issues of performance, clarity and usability. We anticipate that these experiences will lead us to revise some significant details prior to full-scale development, but we expect that the gross features of the architecture will survive.

7 Glossary

Various terms have particular meaning in the context of the Gung-Ho project. Some others are standard jargon in some fields but may be unfamiliar to some readers. Definitions are provided here to provide clarity in their use elsewhere in this document.

Application programming interface (API) A collection of specified and documented procedure interfaces, data types and semantics which provide specified functionality. An API enables a programmer to use functionality without being directly concerned how that functionality is implemented. For example, MPI provides an API for message passing. A programmer can use this to communicate between processes without being concerned about how the message passing is implemented.

Direct Addressing An addressing scheme in which the indices of the degrees of freedom within the stencil of an operation can be directly computed, for example using offsets from the index of the central degree of freedom in the stencil. Direct addressing is often simpler and may offer higher performance than indirect addressing, however direct addressing can only be employed for structured meshes.

ESMF Earth System Modelling Framework.

Indirect Addressing An addressing scheme in which the indices of the degrees of freedom within the stencil of an operation are looked up in a neighbour list. Indirect addressing is required if unstructured meshes are to be employed and provides flexibility in parallelisation, but may have inferior memory access performance to directly addressed memory access.

Mesh The topological and geometric object over which the simulation occurs.

NWP Numerical Weather Prediction.

Topological Entity A topological component of the mesh, for example a vertex, edge or face.

References

© Joint Crown and NERC Copyright 2013

C.J. Cotter and J. Shipton. Mixed finite elements for numerical weather prediction. *Journal of Computational Physics*, 231(21):7076 – 7091, 2012. doi: 10.1016/j.jcp.2012.05.020.

Maff Glover. Gung-ho technical infrastructure: Requirements summary. 2013.

GungHo Team. Grids workpackage: Phase i report. 2013.

A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, 4(4):283 – 295, 2009.

A.E. MacDonald, J. Middlecoff, T. Henderson, and J.L. Lee. A general method for modeling on irregular grids. *International Journal of High Performance Computing Applications*, 25(4):392–403, 2011. doi: 10.1177/1094342010385019.

G. R. Mudalige, I. Reguly, M. B. Giles, C. Bertolli, and P. H. J. Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing conference (InPar '12)*, 2012.

Stephen M. Pickles and Andrew R. Porter. Developing nemo for large multicore scalar systems: Final report of the dcse nemo project. 2012.

© Joint Crown and NERC Copyright 2013

Met Office
FitzRoy Road, Exeter
Devon, EX1 3PB
UK

Tel: 0870 900 0100
Fax: 0870 900 5050
enquiries@metoffice.gov.uk
www.metoffice.gov.uk

Produced with the Met Office LaTeX template v2.0.0(Modified for Gung Ho)